MODULE 4

Session 1 of 4

ASP.NET Core Fundamentals

.NET Runtime & the Request Pipeline

Program.cs

```
var builder =
  WebApplication

  .CreateBuilder(args);


builder.Services
  .AddControllers();


var app = builder.Build();


app.UseRouting();
app.UseAuthorization();
app.MapControllers();
app.Run();
```

Session Overview

By the end of this session, you will be able to:

01 Explain the .NET runtime

What happens between pressing Run and seeing a response in the browser

02 Understand JIT compilation

Why C# compiles to IL first and when the JIT penalty actually matters

03 Order middleware correctly

Why pipeline order is more impactful than any other decision in Program.cs

04 Distinguish Use() from Run()

The one line that separates pipeline continuation from silent termination

05 Diagnose ordering bugs

Read Program.cs and identify middleware mistakes before they hit production

06 Build the TMS pipeline

Implement a correct, production-ready request pipeline for the TMS API

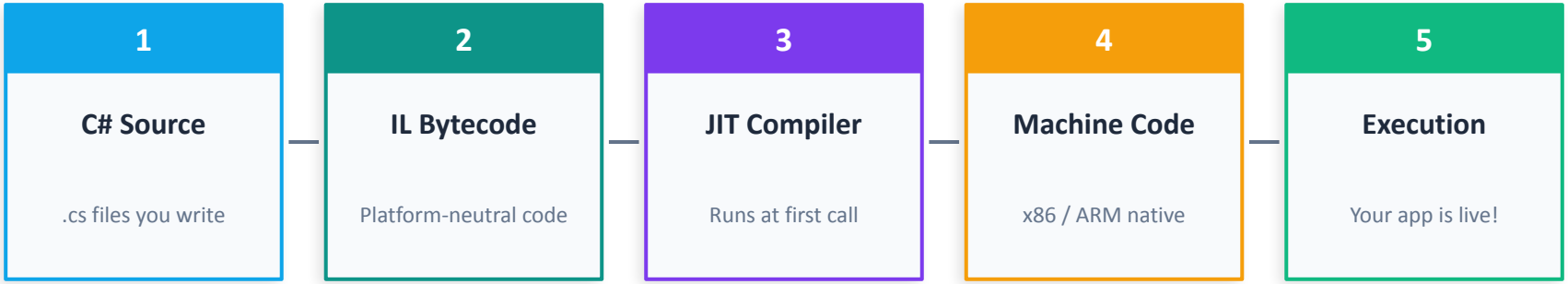
OPENING QUESTION

When you press Run what happens first?

Where does your code actually start executing?

Discuss with the person next to you 2 minutes

The Journey from Code to Execution



Key Insight

Your C# code is never executed directly. It compiles to IL (Intermediate Language) a portable bytecode. The JIT compiler translates IL to native machine code the first time each method runs. This means the same .NET app runs on Windows, Linux, macOS, and ARM without recompilation.

JIT: One-time Cost Per Method

First call compiles the method. Every call after uses cached native code. The cost is startup, not per-request.

Why IL and Not Direct Machine Code?

Compile to x86 and the app won't run on ARM. IL is platform-neutral the runtime on each machine handles the final compilation step.

JIT Compilation Debug vs Release

First Request vs. Subsequent

Request 1 — 52ms

JIT compiles (50ms) + runs (2ms)

Request 2 — 2ms

Cached machine code instant!

Request 3 — 2ms

Cached machine code instant!

The JIT penalty is a one-time startup cost not per-request.

Debug vs Release Mode

DEBUG	RELEASE
· Minimal JIT optimization	· Aggressive JIT optimization
· Preserves debuggability	· Inlines & eliminates code
· Step through every line	· Cannot step through optimized code
· Slower execution speed	· Full production performance

Always benchmark in Release · Always debug in Debug

The Rule

If your app feels slow in development, check: are you running Debug? Debug trades performance for debuggability. Switch to Release mode before benchmarking. If the first production request is slow, that is JIT warm-up completely normal.

Every Request Walks Through a Chain

Before your code runs, ASP.NET Core passes the request through a series of middleware components each handling one concern.

REQUEST flows down →

← RESPONSE flows back up

TMS Scenario Yared clicks 'View Enrollments'

① Logging starts a Stopwatch and adds an X-Correlation-Id header

② Exception Handler arms itself it will catch any crash in the layers below

③ Routing matches GET /api/enrollments to EnrollmentsController.GetAll()

④ Authentication verifies Yared's JWT token confirmed: instructor role

⑤ Authorization confirms the Instructor role can access this endpoint allowed

⑥ Endpoint runs enrollments fetched, 200 OK returned

⑦ Response travels back through every layer in reverse logging records timing

Use() vs Run() One Line Makes All the Difference

app.Use() Continues Pipeline

Program.cs

```
app.Use(async (context, next) =>
{
  // Code BEFORE next middleware
  await next.Invoke(context);  // ← KEY LINE
  // Code AFTER next middleware
});
```

Calls next.Invoke() to hand control to the next component. Runs code BEFORE and AFTER the rest of the pipeline.

Pipeline continues to next middleware

app.Run() TERMINATES Pipeline

Program.cs

```
app.Run(async context =>
{
  // Process and STOP here
  await context.Response
    .WriteAsync("OK");
  // No next.Invoke()  pipeline ENDS
});
```

No next.Invoke() call. Pipeline ends completely. Every middleware registered AFTER this line is dead code.

Pipeline terminated nothing after this runs

The Run() Trap Silent. No Errors. Everything Dead.

No error message. No warning. The app starts but every API endpoint silently returns 404.

BROKEN Can you spot the bug?

```
var app = builder.Build();


// Run() TERMINATES the pipeline here
app.Run(async context =>
  await context.Response.WriteAsync("OK"));


// DEAD CODE  Run() already killed the pipeline!
app.MapGet("/api/enrollments", () => ...);
app.MapGet("/api/courses", () => ...);
app.MapGet("/api/students", () => ...);
```

What actually happens:

- ✓ App starts no startup error at all
- ✓ GET / responds with 'OK'
- ✗ GET /api/enrollments → 404
- ✗ GET /api/courses → 404
- ✗ GET /api/students → 404
- ✗ Every TMS endpoint is unreachable

FIX: Move app.Run() to the very last line. All MapGet(), MapControllers() calls must appear before it.

Middleware Order The Correct Pipeline

Each layer depends on work done by the one above it. The rule: ask 'What does this middleware NEED from earlier in the pipeline?'

1	`Custom Logging Middleware`	Wraps everything measures total time, adds Correlation ID to all logs
2	`UseExceptionHandler()`	Catches exceptions from any layer below must be near the top
3	`UseHttpsRedirection()`	Forces secure connection before any real work starts
4	`UseRouting()`	Determines which endpoint matches CORS and Auth depend on this
5	`UseCors()`	After routing so preflight OPTIONS requests get a valid response
6	`UseAuthentication()`	Determines WHO the user is Authorization depends on this
7	`UseAuthorization()`	Determines WHAT the user can access needs Authentication to have run first
8	`MapControllers()  /  app.Run()`	Your actual code runs last, after every security check has passed

Middleware Order Real-World Mistakes

Same spine as M4-Session-1-Presentation wrong order has real cost, not style.

Scenario 1 Authentication before routing

Program.cs

```
app.UseAuthentication();
// runs on EVERY request

app.UseRouting();
// including non-existent routes
```

Result: Auth runs for undefined routes wasted CPU; callers may see Unauthorized instead of Not Found.

Scenario 2 Rate limiting after the handler

Program.cs

```
app.MapGet("/api/grades", () =>
    db.Query(...));  // expensive

app.UseRateLimiting();  // too late
```

Result: The expensive work already executed throttling cannot protect capacity.

Contrast with previous slide

Canonical order puts routing before auth so 404s skip auth, and cross-cutting limits before endpoints so work is gated.

Lab 1

The Blind Server

Pipeline Repair Middleware Ordering

1. An endpoint returns sensitive GPA data to anonymous users find out why
2. Read the legacy Program.cs and identify the ordering mistake
3. Rewrite the pipeline in the correct sequence
4. Verify the endpoint now correctly rejects unauthenticated requests

Full steps in M4-Lab-Workbook.md · Exercise 1

Session 1 Summary What You Learned

JIT Compilation	C# → IL → machine code at runtime. Compiles once per method, then cached. Debug mode slows it deliberately.
Debug vs Release	Always benchmark in Release. Debug preserves debuggability; Release applies aggressive JIT optimization.
Request Pipeline	Every request flows through middleware in order down on the way in, back up on the way out.
Use() vs Run()	Use() calls next.Invoke() to continue. Run() terminates. Forgetting next silently kills everything after it.
The Run() Trap	app.Run() before endpoint mapping = dead endpoints. No error, no warning. Always put it last.
Middleware Order	Logging → Exception → Routing → CORS → Auth → Authz → Endpoints. Each layer needs the one before it.

What This Looks Like in an Interview

Companies want people who understand the runtime model, not just syntax. Senior engineers review middleware order. If you can explain why authentication must come before authorization, you demonstrate architectural thinking not just tutorial-following.

Add routing: why auth runs after UseRouting() in the canonical chain tie both sentences together in one answer.

Coming Up in Session 2

Dependency Injection & Service Lifetimes

You'll learn why every service needs a lifetime, how the DI container wires your app together, and the captive dependency bug the most famous silent data-corruption bug in ASP.NET Core.

The most critical topic in M4